CIM-Tile Architecture and Simulator

Mahdi Zahedi

August 28, 2023

This document is part of my Ph.D. thesis titled "Computation-in-Memory from Application-Specific to Programmable Designs". This helps the user of this code to understand the concept first and then how to use it. For further information, please contact me.

1 CIM Tile Initial Structure

In our perspective, a CIM tile can be seen as an off-/on-chip component from the CPU. In this way, CIM tiles are not integrated into the memory hierarchy of the system and are allocated into different address spaces. This simplifies system integration (in the future) since the designer is not concerned with memory coherency. In this section, we focus on the structure of a singel CIM tile and how it is organized. First, we describe the initial version and its limitation. To address that, we explain the extended version. Finally, we elaborate on possible ways to perform pipelining inside a tile.

1.1 Tile Overview

Figure 1 depicts the architecture of the CIM-tile, including digital and analog components as well as control and data signals. The operations that can be executed on the crossbar are divided into two main categories: 1) write and 2) read/computational operations. The computational operations include *addition, multiplication*, and *logical operations*. In the following, we will describe our tile architecture and its main modules considering these two categories: (the discussion of the control signals is left to the subsequent section).

1. **Write operation**
Before doing any computation on the crossbar, the memristors in the crossbar(s) have to be programmed. In the case of the 1T1R crossbar structure, the memristors located in the same row can be programmed in parallel. Otherwise, sequential programming is required. In order to write data to the memristor crossbar, we have to specify the location in the crossbar (based on the index of row and column) where the data has to be written to. Therefore, three registers are employed to capture this information. 1) The data itself has to be written to the *Write Data (WD)* register, whose length depends on the width of the crossbar as well as the number of levels supported by the memristor cells. Considering endurance issues and potential energy savings, it is not always necessary to write data to all the array columns. 2) For this purpose, the *Write Data Select (WDS)* register is used to select which columns should be activated. This is especially relevant when considering implementing a write-verify operation. 3) Finally, the *Row Select (RS)* register is employed to activate the row in which data has to be written. In this version, the data required to fill these registers is embedded into the institutions. A more detailed description is presented in the following.

Voltages that have to be applied to the crossbar depend on the crossbar technology, and they are usually different than the voltage used for the digital part of the system. Therefore, we need a device to convert the information from the digital to analog domain called Digital Input Modular (DIM). Selecting a row requires that two different voltage levels have to be provided for both the source and world-line of the target row as depicted in Figure **??**(b), which means two DIMs (*Source/Gate DIM*) are required to drive both of them. Therefore, considering one DIM for the crossbar columns (*Write DIM*), we need three DIMs in this architecture in total. Based on the operation and the data stored in the RS and WD registers, DIMs can apply proper voltage levels to the crossbar. In addition, the data in the WDS register is used by the *Mask unit* to prevent extra switches for the cells that the data has not to be written into them.

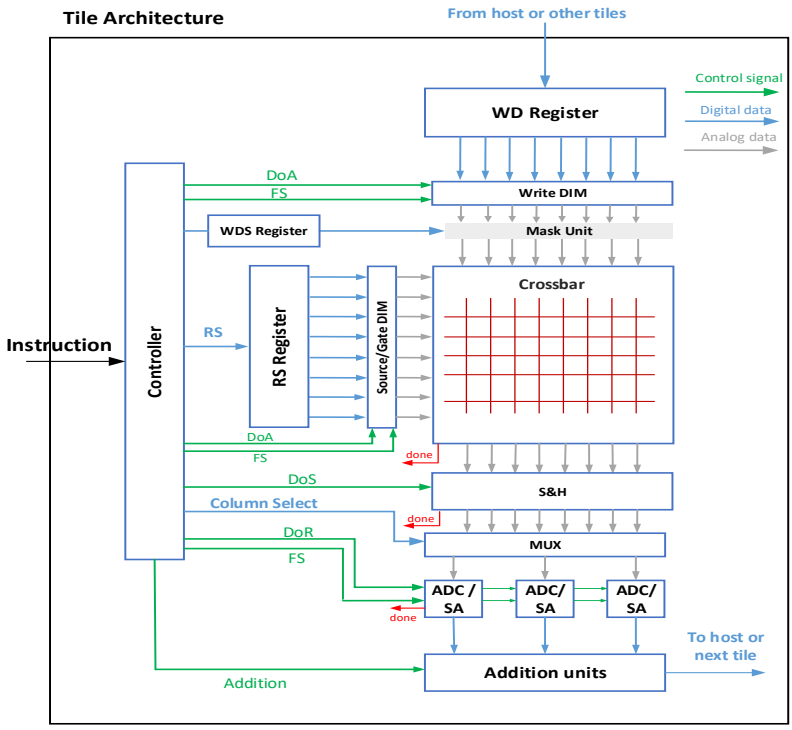

Figure 1: The overall tile architecture.

2. **Read and computational operations**
In this category, the operations generate an output and it has to be read by the periphery circuits in the architecture. The generated output can be the outcome of either a normal memory read or computational operation. In contrast to the write operation, there is no need to fill the WD and WDS registers. The RS register is again used for row activation. However, among computation operations, matrix-matrix multiplication (MMM) is different than others in the sense that the RS not only has to indicate the active rows, but also can be considered as the data for one of the matrices. When the operation is performed inside the crossbar, the generated analog output has to be captured by the *Sample & Hold* (S&H) unit. This allows for a clear separation of the execution within the array and the read-out circuitry, which can be used for the pipelining of the system.

When the S&H module has captured the result from the array, the ADCs (or the sense amplifiers) can be used to convert the analog results into the digital domain. Since ADCs consume much energy and area, usually it is not possible to allocate one ADC per column. Therefore, we need analog multiplexers to share several columns with one ADC. Besides, certain high-level operations, e.g., the integer MMM, require additional processing steps and these are performed in the *Addition Units* where the design utilizes minimum size adder to impose as less latency/power as possible to the system. The design considers technology/circuit/application-driven restrictions such as the maximum number of active crossbar rows, number of ADCs, and datatype size. When other (high-level) operations are needed in the future, this unit can be altered or substituted with others.

1.2 CIM tile (Micro) instruction set architecture

As discussed before, a complex sequence of steps needs to be performed in the CIM tile that can be different depending on the (higher-level) CIM tile operation, e.g., read/write, dot-matrix multiplication, Boolean operations, and integer matrix-matrix multiplication. Similar to the concept of microcode, we introduce an (micro) instruction-set architecture (ISA) for our CIM tile that would allow for different schedules for different CIM tile operations. Despite conventional instruction, these newly-defined instructions expose the hardware of a CIM-Tile to the programmer to provide high flexibility in the execution of a variety of operations. The "Controller" in Figure 1 is responsible for translating these

Table 1: List of initial instructions.

Instruction	Semantic	Operand
Row select	RS	data to fill RS register
Write data	WD	data to fill WD register
Write data select	WDS	data to fill WDS register
Function select	FS	data to configure the drivers/ADCs
Do array	DoA	-
Do sample	DoS	-
Columns select	CS	data for the select of MUX
Do read	DoR	-

instructions to the actual control signals (highlighted in green). The list of instructions is presented in Table 1. In the following, we discuss each instruction:

- *Row Select (RS):* The RS instruction is responsible for setting up the RS register (see Figure 1) that is subsequently used to correctly control the source and gate drivers to provide the right voltage levels for the crossbar. At this moment, the number of bits to set the RS register is as large as the height of the crossbar (which means the input precision is 1 bit). As the size of the crossbar increases, this is impractical for hardware implementations and will be addressed in the extended version of instructions. However, for simulation purposes, the impact is negligible and actually allows us to investigate the utilization patterns for the RS register.

- *Write Data (WD):* The WD instruction is responsible for setting up the WD register that is used to write data into the crossbar. Similar to the RS instruction and with the same reasoning, the instruction size (excluding the opcode) is as large as the size of the WD register. We envision that this instruction to be replaced when another mechanism is chosen to load data into the crossbar that is more hardware-friendly. For simulation purposes and in the initial version, it is now the only way to load data into the crossbar.

- *Write Data Select (WDS):* The WDS instruction sets up the WDS register to control which bits of the WD register need to be written. This allows for a flexible manner to write data into the crossbar in light of potential endurance issues associated with current-day memristor technologies. It is especially useful when a write-verify operation needs to be performed where the written data into the crossbar is compared with the golden data. With the help of this register, correctly written bits can be masked out, which helps to improve the endurance of the crossbar.

- *Function Select (FS):* The FS instruction is needed for several reasons. Functionally, the DIMs need to be set up differently when writing data into the crossbar, or when reading out data, or performing compute operations in the crossbar. Furthermore, it is envisioned that the future periphery needs additional control signals. These are now conceptually captured in the FS instruction. For example, the control signals needed to control the "Addition units" are more than one, i.e., more than one instruction is needed to control these units. The instructions related to this unit are provided in the extended version.

- *Do Array (DoA):* The DoA instruction is used to actually initiate the DIMs after they have been set up using the RS, WD, and FS instructions. This instruction will have a variable latency depending on the operation that is performed in the crossbar. These delays are specified as parameters in our simulator. In the hardware implementation, this latency (of the crossbar) can be captured by a (programmable) counter or by a 'done' signal issued from the crossbar itself. More importantly, this instruction allows for a clear (conceptual) separation between the different pipeline stages within the CIM tile. This will be discussed later.

- *Do Sample (DoS):* The DoS instruction is used to signal the S&H module to start copying the result from the crossbar into its own internal storage. It must be noted that this module still operates in the analog domain. The introduction of the DoS instruction allows for a conceptual separation of the execute stage (crossbar) and read stage (ADC). After the values are copied into the S&H module, the crossbar can basically be activated by the next DoA instruction.

- *Column Select (CS):* The CS instruction is used to set up the CS register that controls the multiplexers (in the MUX module) in the read stage. In case each column of the crossbar can be associated with its own ADC/SA, there is no need to have the CS instruction. In all other cases, the CS register flexibly controls which column (in the S&H module) is connected to an ADC/SA. The length of the CS instruction is related to the number of columns of the crossbar. For the same reasons as with the RS and WDS (and WD) instructions, it is purposely defined as it is now in this initial version and implemented as such in the simulator to allow for further investigation in the future. It is expected to be optimized or replaced when a hardware implementation is considered.
- *Do Read (DoR):* The DoR instruction initializes the modules "ADC/SA" to start converting the output of the S&H module (via the MUX) into a digital representation. It is expected that multiple iterations of the CS and DoR instructions need to be issued in order to completely read out all the columns of the crossbar. It should be noted that when none of the columns allocated to an ADC/SA are selected, the sensing unit in is not activated. Depending on the complexity of the module, the latency can vary, and thus a 'done' signal is needed (see Figure 1) to signal the end of the readout before the next DoR instruction can be issued. This instruction might also be quite practical when we have ADCs with configurable resolution. The resolution of ADC is directly related to how many times it gets activated for an input signal.

It is important to note that the crossbar, the S&H modules, and the ADC modules (related to the DoA, DoS, DoR instructions, respectively) need to be able to signal to the controller that their operation is finished. It is only after this signal that subsequent instructions can be issued by the controller. If this turns out to be impossible in an actual hardware implementation, we envision the need to set up counters that are initialized to the specific operations to achieve the same functionality. Both approaches are already supported in our simulator.

1.3 Challenges of the initial design

Although the ISA defined before brings programmability and flexibility into the design, they impose significant challenges when it comes to their hardware implementation. In the following, we elaborate on the main limitations of the proposed ISA. Based on that, we present the extension of the CIM-tile structure and its ISA.

- The initial set of instructions embeds all of the data that has to be fed to the crossbar into the instruction. For example, in the case of MMM, while one operand is programmed into the crossbar, the second operand is given to the RS register to be fed to the crossbar as input. This data is embedded into the 'RS' instruction. Similarly, the data that has to be programmed into the crossbar is embedded into the 'WD' instruction. However, for real applications, the data may not be known at compile time and produced during execution. We can consider a deep neural network as an example here, where the input to the intermediate layers is generated during the execution time. Hence, assuming the data is always present in the instruction is unrealistic. Therefore, when designing the instructions set, it should thus be taken into account that compile-time unknown data is fed to the CIM-tile at run-time.
- The CIM-tile would be a part of a micro-architecture. From the system perspective, the data for a CIM-tile is provided either from another Tile or an external component (e.g., CPU). This data is transferred via a databus. In a realistic assumption, the bandwidth of this databus limits the amount of data that can be read per instruction. Since we have not considered any hard constraints on the size of databus, the current design is not conditioned by the fact that the databus might be too small to provide all required data in a single cycle. Hence, a proper mechanism should be defined when interfacing the tile with the outside.
- The instructions generated for an application/kernel should be stored in memory to be executed during run-time. Since the storage of the instructions is assumed to happen at compile time, the instruction memory sets an upper limit for the size of a program. Therefore, it is essential to have a reasonable size for each (micro) instruction to use available instruction memory size more efficiently. Considering the initial version of the instructions, the implementation for some of them, such as row selection, write data, write data selection, and column selection instructions, would be data-intensive. Even a small-sized MMM operation is translated to an instruction size between 0.49 and

Table 2: Instruction file exploration for the Gemm benchmark

#ADC	ADC resolution	File size (MB)	%RDS	%WD/WDS	%CS
8	5	13.75	3.08	0.06	96.8
8	8	1.74	3.44	0.45	95.67
32	5	3.8	11.25	0.21	88.34
32	8	0.5	12.17	1.59	84.66

3.4 MB, depending on the ADC precision and the number of ADCs. Hence, the instruction should be revisited with this aspect in mind.

To get an indication regarding the total instruction file size as well as the contribution of each instruction, we have compiled the 'Gemm' benchmark, consisting of a single matrix-matrix multiplication, preceded by storing the multiplicand matrix in the crossbar. Table 2 shows the contribution of the individual data-intensive instructions to the instruction file size. The exploration is done assuming the crossbar size of 256×256. As we discussed, 'RDS', 'WD', 'WDS', and 'CS' are the main contributors to the instruction file size. While all of the mentioned instructions implementations should be modified in order to attain hardware feasibility, the main data reduction can thus be expected in the change of the CS instruction implementation. It should be noted that the sequence of instruction depends on many parameters, including the number of ADC and their resolution. We explain this later. This information provides insight to steer toward the extended CIM-tile structure and (micro) ISA.

2 CIM Tile Extended Structure

2.1 Tile Overview

The overall CIM-tile architecture presented throughout this section is depicted in Figure 2. The CIM-tile is expected to interact with external devices and therefore, a clear interface needs to be defined. The interface comprises two input buffers (Write-Data (WD) and Row-Data (RD)) and an output buffer. This allows independent CIM-tile operation without needing an external controller knowledgeable of the CIM-tile internals. Specifically, the external components can be agnostic about the CIM-Tile instructions and their status. These buffers are explained in the following:

- *WD buffer:*

  This buffer serves as intermediate storage to alleviate the timing constraints of sending data to the CIM-tile when the (bus) bandwidth is insufficient to transfer all required data in one shot - i.e., to the WD register. Therefore, each WD buffer row corresponds to a chunk of data received from the outside controller. This data is going to be programmed into the crossbar, ultimately. The WD buffer has been sized such that an entire (array) row of data can be stored inside it. Consequently, after transfer to the WD register during a write operation, the WD buffer is ready to receive the next set of data. There would be instructions to stream the data out of this buffer. We will discuss this in the following subsection. Using this buffer, a mismatch between crossbar size and bus width can be handled. The two main architectural parameters that impact this buffer are 1) the bus width provides input to the buffer from outside and 2) the number of crossbar columns. Based on that, there would be two scenarios:

  In the best scenario, the bandwidth is large enough to provide all the required data to be written into one row of the crossbar in a single cycle. This means the data is written into this buffer in a single cycle, and then the tile controller copies the data to the WD register for the next write operation. However, in a more realistic scenario, the bandwidth is small to be able to transfer all the data in one cycle. Therefore, a new design parameter is introduced for the WD buffer size. The buffer size can be either 1) equal to the available bandwidth (minimum size buffer) or 2) equal to the crossbar width to save all the information for a single row. In the following, we discuss these two cases.

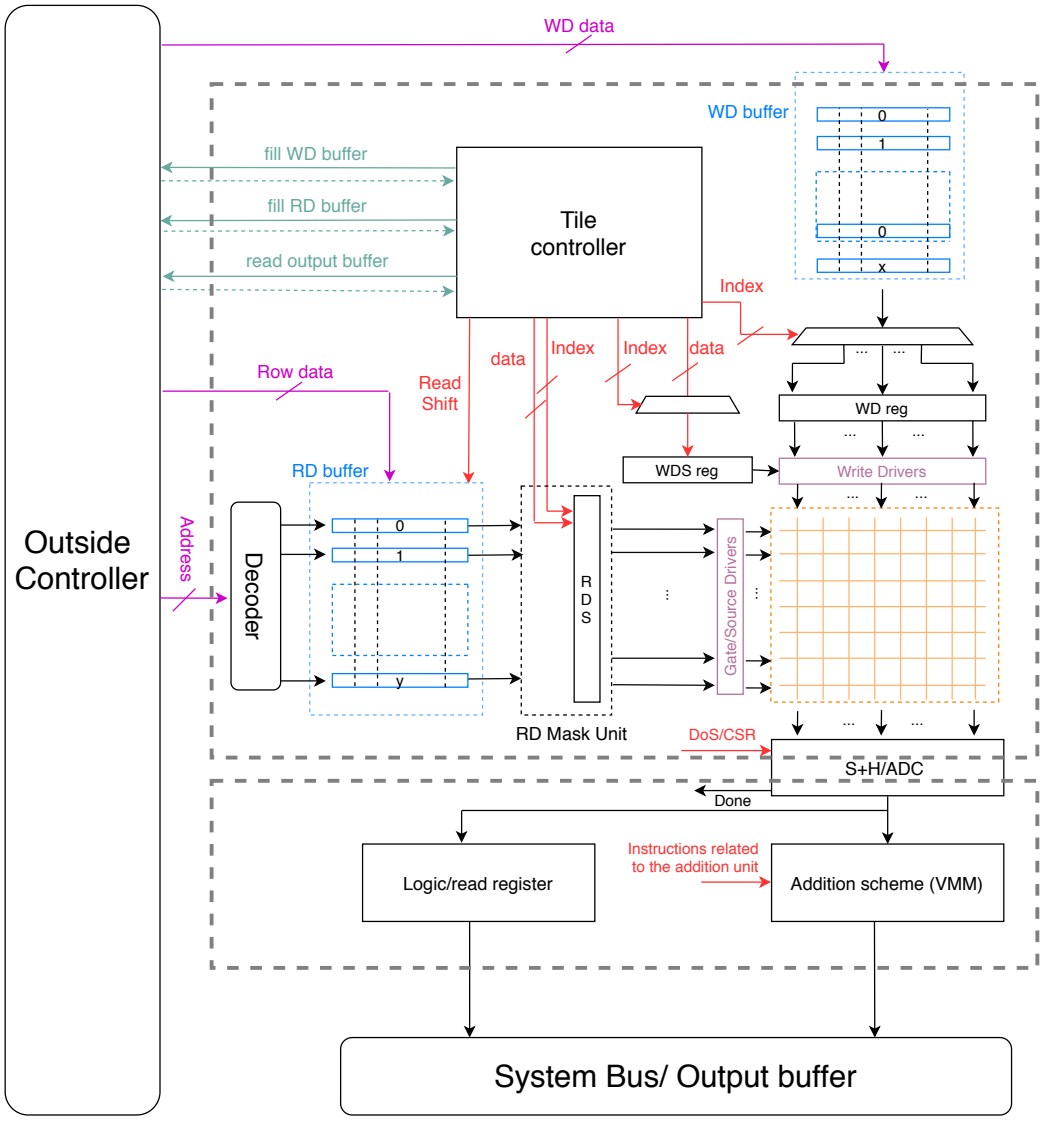

Figure 2: Overview of CIM-tile extended structure.

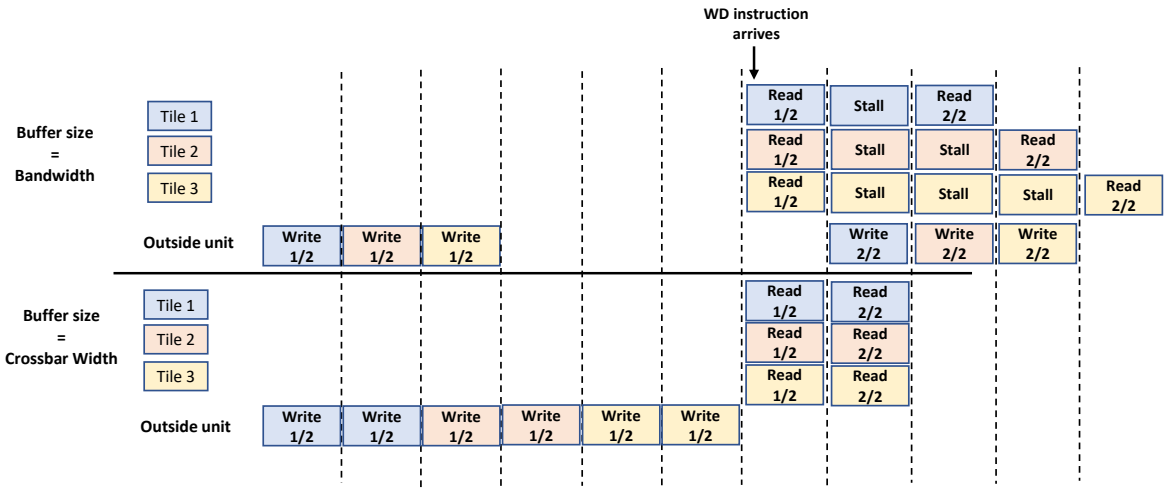

Figure 3: Execution flow for three tiles considering two WD buffer sizes. In this example, all tiles arrive at the WD instruction at cycle 6.

The main advantage of having a minimum size buffer is the lower area cost. However, this leads to timing constraints for filling the buffer. To clarify, consider an example where a crossbar width is 256 columns (each column one bit), and the bandwidth is 128 bits. During the first cycle, the outside unit provides the first 128-bit to the WD buffer. During the second cycle, this data will be put in the correct position of the WD register by the tile controller. In the third cycle, the outside unit will write the second 128-bit to the WD buffer. During the fourth cycle, the second 128-bit can be put in the WD register, meaning the register is full, and the tile can thus perform its write operation. Now, we consider the case of a WD buffer of size equal to the width of the crossbar (so 256-bit). We still need to spend two cycles for outside to transfer the data to the RD buffer. However, this can be two subsequent cycles as the CIM-tile does not need to process the first half. Hence, the larger the WD buffer is, the fewer timing constraints are between the outside unit and the tile controller.

Figure 3 illustrates the aforementioned example where there are three different tiles. In this example, the WD instruction arrives at the same time for all three tiles. Despite the possible performance overhead of the first case (buffer size equal to the bandwidth), the major challenge is the synchronizations between the outside and the CIM-tile. The actual performance impact is dependent on a number of factors, such as the number of tiles serviced by the outside unit, the application (Number of write operations and their position in the program) and the available bandwidth for writing to the WD buffer, and the cycle latencies of other pipeline stages of the tile. Hence, in our design, we considered the WD buffer to be equal to the crossbar width. Now that the buffer size has been determined, two implementations of the buffer can be considered.

1) In the first implementation, a buffer based on the FIFO shift register may be employed, of which the width is equal to the bandwidth for writing to the buffer and the height is such that the product of width and height equal the size of the crossbar width. This solution has two drawbacks. First, as only the topmost row of the buffer can be accessed by the outside unit, a fixed number of shifts is required for the first set of data to arrive at the bottom of the buffer. Secondly, shifting the data through the entire register leads to extra switches in the buffer, leading to energy usage overhead.

2) In the second implementation, a counter that increments on a write from the outside unit, decrement on a read from the tile, and remains constant when both happen during the same cycle is placed. It can be ensured the data from the outside is written to the bottom-most free row of the buffer. The data present in the filled rows of the buffer is shifted downwards one row when the tile reads a row from the buffer. An obvious drawback of this third solution is the extra hardware that is required to allow for the individually addressable rows of the buffer and the simultaneous reading and writing. Figure 4 shows the design of this buffer.

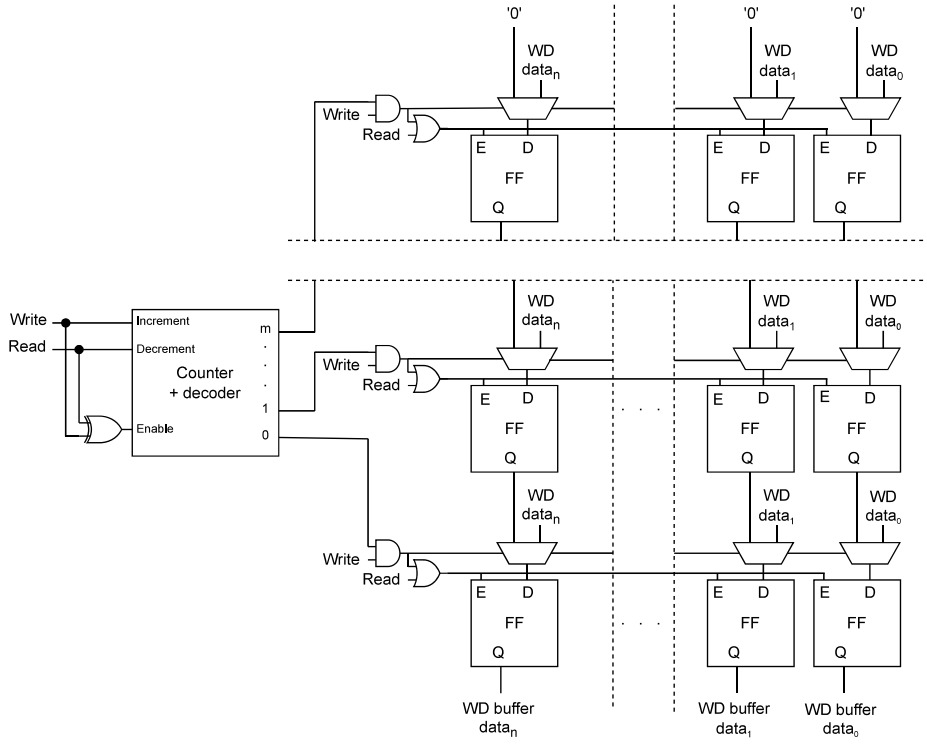

Figure 4: Implementation of the WD buffer.

- *RD buffer:*

  In the case of computation operations, the RD buffer holds one operand of operations and feeds it to the crossbar. Since the crossbar drivers (DAC) have limited resolutions (usually one bit), the data must be split and given to the driver in several cycles. As this buffer feeds data into the crossbar bit-by-bit, starting from LSB to MSB during a Matrix-Matrix Multiplication (MMM) operation, this buffer is implemented using a parallel-in-serial-out (PISO) register per crossbar row. By having the width of the buffer sized to fit the maximum supported datatype size (e.g., 32-bit), the buffer only has to be filled once for each of the element dot-product operations. The height of the buffer is equal to the number of crossbar rows. After shifting the MSB out of the buffer, the next set of data can be loaded into the buffer to reduce the overhead of data transfer. It should be noted that after each shift to the right, the output of the buffer is written back to the input. This is needed when computing on a datatype size lower than the one the system is designed for. This relates to the 'Addition Scheme' and how this periphery unit works.

  In some scenarios, the data for the RD buffer is loaded from external memory (e.g., DRAM or Flash). This data is a vector of elements, and in each cycle, one (or a few) bit(s) should be given to the crossbar (due to DAC resolution). In case there is no RD buffer, when this (vector) data is loaded from external memory, it needs to be processed to extract and combined the bits required from each element and send them to the crossbar. Hence, it needs temporary storage as well as extra processing. This indicates another advantage of this RD buffer which makes the outside completely unaware of the tile limitations (e.g., DAC resolution) and removes the extra processing and the temporary storage.

- *Output buffer:*

  This buffer is employed only for temporarily storing the result data until the data has been transferred to other tiles or processors. As the output of the crossbar should be stored in this buffer, its size determines the amount of parallel computation that can be performed. Without this buffer, the outside world should be synchronized with the execution of instructions in the tile, which can increase the complexity on both sides. The output buffer also has to be sized properly. The size of the output buffer depends on the datatype size. Table 3 shows the size of each element generated

Table 3: Required output buffer sizes assuming 256×256 crossbar and 1 bit per each memristor.

Datatype size	Element size (bit)	Output buffer (bit)
32	72	576
16	40	640
8	24	768
4	16	1024
2	12	1536
1	8	2048

by a crossbar and the required output buffer size based on different datatype sizes. This number is brought for MMM operation and the crossbar and assuming both operands have the same datatype size. While the result of a single element for a smaller datatype is smaller than for the maximum datatype size, more elements can be stored in the crossbar, and thus more computation can be performed. Therefore, based on the datatype sizes intended to support, a designer has to place a proper output buffer. In addition, to flexibility support different datatype sizes, more consideration has to be taken into account.

In addition to the buffers, other digital components (depicted in Figure 2) are:
Tile registers: Same as the initial version, (i) the *Write Data (WD)* register contains the data that has to be written into the crossbar. The data loads from the *WD* buffer in several steps depending on the WD buffer row size. The register length depends on the width of the crossbar as well as the number of levels supported by the memristor cells. However, due to the flexibility attained by our instructions, we can opt to only partially fill the register if the kernel does not intend to write into all of the columns. (ii) The *Write Data Select (WDS)* register indicates to the write drivers which columns should be written to. The data for this register is embedded in its instruction to provide more flexibility for kernels. Finally, (iii) the *Row Data Select (RDS)* register is employed for the activation of crossbar rows and used for all operations, including write, read, logic, and VMM. In addition, in the case of VMM, if the ADC resolution does not support the activation of all the rows, this register is used to activate batches of rows in several steps. Similar to the *WDS* register, the data for this register is embedded into its instruction. To clarify more, the data in the RDS register shows which region of the crossbar should be active, while the RD buffer stores the data for one operand of compute operations.

The other two main components are the tile controller and the 'Addition Scheme'. The controller will be discussed in this chapter, but the discussion regarding the addition unit is left out of this document.

2.2 CIM Tile (Micro) Instruction Set Architecture

To execute a kernel on the tile, a complex sequence of steps has to be carried out considering different operations, the patterns of column and row selections, datatype sizes, and read-out circuitry specifications. An extension of in-memory (micro) instructions is defined with the objective of keeping the hardware simple and generic by moving the complexity to the compiler, reducing the instruction file size, and maintaining high flexibility. The list of instructions is presented in Table 4. The instructions are explained shortly in the following.

- Row Data (***RDxx***): The first group of instructions is related to the crossbar rows.

  Row Data Selection: In the initial version of this instruction, we dedicated one bit for each row of the crossbar to indicate whether that row should be activated or not. This causes an issue in terms of the instruction size. To address this, two solutions can be provided. In the first solution, which is called row-wise row selection, we only pass the index of a few rows into the instruction. In case the instruction allows us to embed 24 bits of data into it (excluding opcode) and the crossbar size of 256, we can only address three rows at a time ($3 = 24/log2(256)$). Hence, multiple of this instruction should be used to activate more rows. The advantage of this is activating any random rows in case the application does not follow any pattern. However, this solution would be inefficient for selecting a large number of rows as the instruction count will scale linearly with the number of

Table 4: Overview of the new ISA

Opcode	Op 1	Op 2	Function description
RDSb	Index	Mask	Place 'Mask' into RDS reg at 'Index'
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift RD buffer contents
WDb	Index		Copy data to for WD buffer to WD reg. at 'Index'
WDSb	Index	Mask	Place 'Mask' into WDS reg at 'Index'
WDSc			Clear the entire WDS register
WDSs			Set the entire WDS register
FS	Function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	Index	Activation	Select column to be read by ADC
DoR			Activate the ADC
jal	Address		Jump to 'Address' and store PC.
jr			Jump to the PC stored in return reg.
BNE			Branch to PC stored in branch reg.
LS			Indicate the last section of rows to reads
IADD			Activate third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Selection		Select adders for addition between ADCs
CB			Copy the result of addition between ADC to output buffer

rows to be selected. In addition, the implementation of this row-wise solution would lead to multiple large decoders.

The second solution of row selection aims to group the rows and indicate which rows are activated within a specific row. This solution is called block-wise row selection. As an example, the crossbar with 256 rows can be grouped into 16 parts; each contains 16 rows. The instruction first identifies the group index ($4bits = log2(16)$) and then provides 16 bits of data for each group. We refer to this instruction as **RDSb**. This solution is more efficient if consecutive rows have to be selected. For example, if only rows from index 0 to 16 should be selected, this can be done by clearing the RDS register and executing a single RDSb instruction. To allow for clearing the register, a new instruction is introduced; RDS-clear (**RDSc**). In case most rows have to be activated, we can set the entire RDS register and use RDSb instruction to deselect the rows that should not be activated. To enable setting the entire RDS register, another instruction is introduced; RDS-set (**RDSs**). Being able to select and deselect the rows in the crossbar is quite essential, especially when the precision of ADC is not enough to activate the entire crossbar rows. In this case, we use this instruction to split the rows into groups and activate them sequentially. It should be noted that ADC precision is an important parameter, and it even has a consequence on the program's size.

Finally, **RDsh** will shift the *RD* buffer to the right to present the next bit for a new compute operation. This instruction is quite useful, especially when we are dealing with datatype sizes smaller than what the system is designed for. When the datatype size in the crossbar is smaller than what a system is designed for, the same input data has to be given to the crossbar for several iterations due to resource limitations in the periphery. This is why, after each shift, the output of the RD buffer is written back to its input. Figure 5 depicts the state of RD buffer after each shift. Since the data size (highlighted by blue) is half the system datatype size, the same data has to be given two times to the crossbar. However, at time step T4, we need to shift the data 4 times to the right to position it correctly. This is done by executing 4 RDsh instructions in a sequence. Employing this instruction, we can move around the data in the RD buffer with high flexibility. Figure 6 illustrates the overview of the underlying hardware for all the instructions related to the row data.

- Write Data (**WDxx**): The **WDb** instruction indicates that the data present in the last register of *WD* buffer has to be moved to a section of *WD* register determined in the instruction by 'index'. If we only want to program part of a crossbar row, this instruction helps us to navigate this information from the WD buffer to the correct location in the WD register. Like row selection, **WDSb** instructions load the data to the *WDS* register. Similar reasoning as for the RDSb solution applies to the write data selection, leading to a block-wise solution for the WDS instructions as well, introducing two new instructions to clear and set the entire WDS register; Write-Data-Select-clear

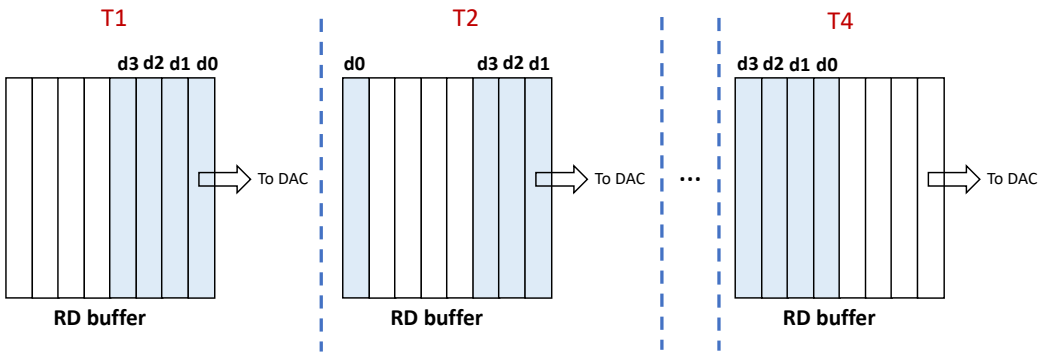

Figure 5: The state of RD buffer when the datatype is smaller than the maximum supported datatype size of the system.

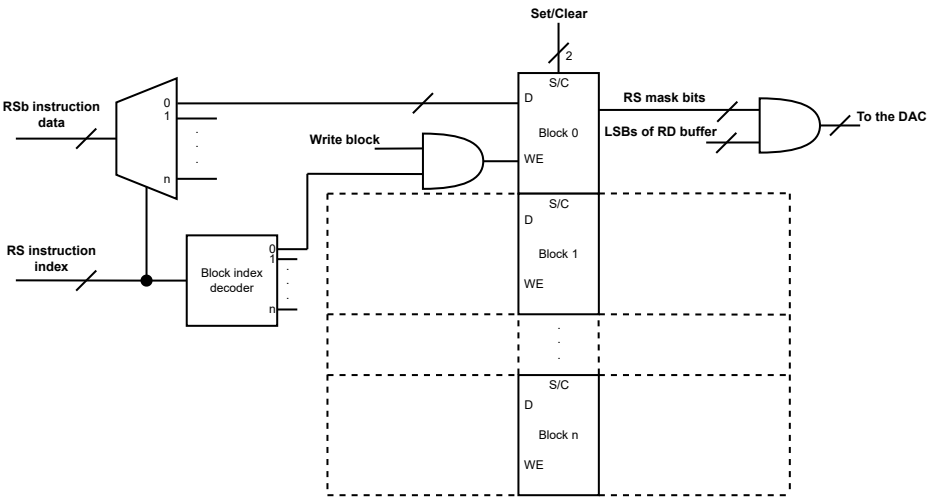

Figure 6: Instruction relates to row data and their underlying hardware.

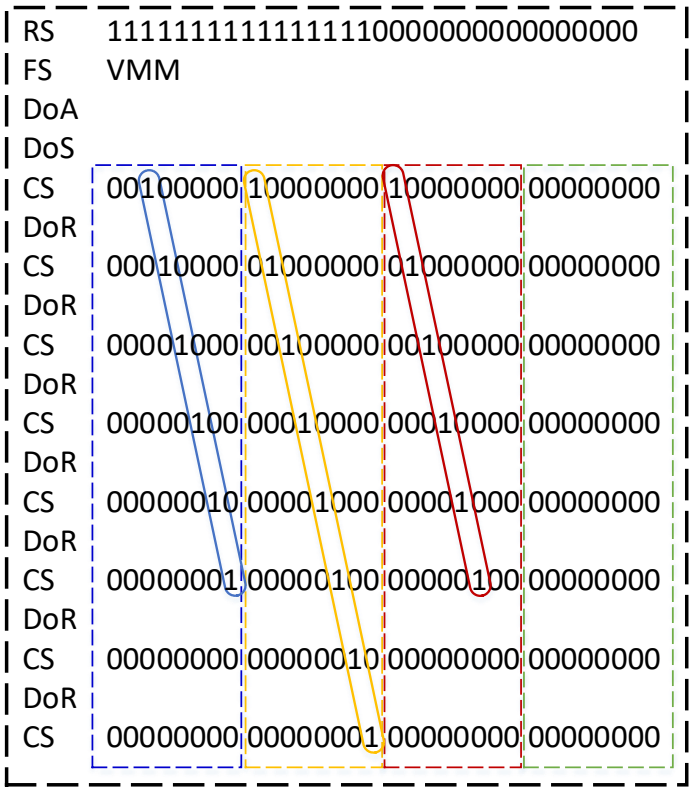

Figure 7: An example of instructions generated by the compiler using the initial version of the CS instruction.

(**WDSc**) and Write-Data-Select-set (**WDSs**), respectively.

- Crossbar (***FS, DoA***): To operate on the crossbar, first, we need to configure the drivers to provide proper voltage levels based on the desired operation. This is done using the Function Select (*FS*) instruction. This instruction is also used to configure the read-out circuitry as well as bypassing the *RD* buffer data in the case of read/write/logical operations. By decoding the *DoA* instruction, drivers are activated, and the operation is started on the crossbar.

- Read Out (***DoS, CS, DoR***): Due to the overhead of Analog-to-Digital Converters (ADCs), they need to be shared between several columns, which translates to the necessity for a *Sample-and-Hold* unit to save the crossbar's output. The *DoS* instruction activates this unit at the right time. The data can be sampled when the second stage is already done with the prior sampled data.

  The initial implementation of **CS** offers large flexibility at the cost of huge instruction size. Figure 7 depicts how this initial version works. In this example, the compiler generates the code for a small crossbar size of 32 $\times$ 32 with 4 ADCs. This means each ADC shares 8 columns. In each cycle, one column from each group is read by an ADC. Due to the instruction size and its frequent recurrence, column select instruction makes up for about 88% of the data in the benchmark. To address this, in the new version of CS instruction, we pass a single index in the CS instruction, which will set all ADCs to that specific index and use a set of activation bits to indicate whether an ADC should read the column at that index. Considering the above example, the size of CS instruction (excluding opcode) is reduced from 32 bits down to 7 bits ($log2(8) + 4$). Finally, similar to the initial version **DoR** instruction activates the conversion.

- Jump instructions (***jal, jr, BNE***): As the read stage often performs identical sets of instructions (e.g., when performing a MMM, the same set of columns has to be read for every dot-product operation), a jump instruction is introduced to save a large portion of the instruction file size. Similarly to MIPS, jump-and-link (**jal**) and jump-register (**jr**) instructions are introduced, allowing

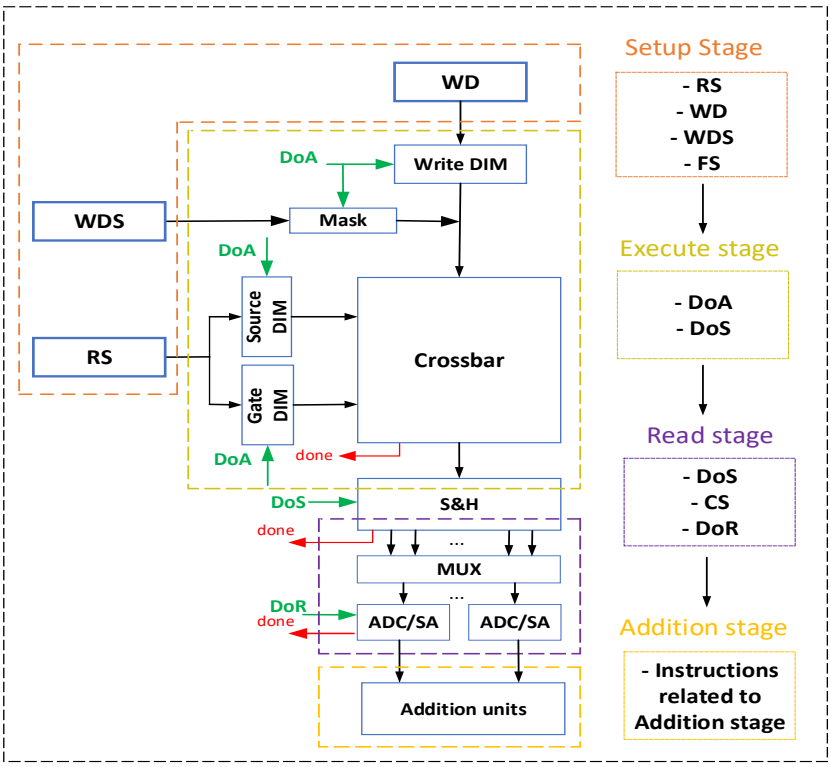

Figure 8: Pipelining of the crossbar CIM-tile.

for re-using the same block of instructions for every read. The 'jal' instruction stores the current program counter and jumps to the set of instructions (often readout). In order to correctly resume the instructions after branching, a 'jr' instruction should be introduced, which allows for returning to the program counter stored by the 'jal' instruction. Branch Not Equal (**BNE**) instruction is used for write verification. Due to the memristor non-idealities, sometimes we need to verify what we programmed into it to ensure its correctness. This instruction performs this task for us and jumps back to repeat the programming if the operation is not verified.

- Addition unit (***LS, IADD, CP, AS, CB***): According to the structure of the addition unit, which will be discussed in Chapter **??**, in the case of low ADC resolution, the *LS* instruction indicates the last section of rows (multiplier) that are activated. The *IADD* instruction performs an addition between different bits of the multiplier. If a number is distributed over more than one ADC, *AS* instructions carry out the addition between the results taken per ADC. Finally, *CP* and *CB* load the result to the output buffer obtained either per ADC or between ADCs, respectively.

3 Pipelining

In order to improve the performance, the CIM-tile can break into a few stages which can be active in parallel. This means that the instructions associated with them can be executed in parallel, which leads to higher performance. In the following, we elaborate more on four possible stages in the tile (indicated by different colors in Figure 8).

1. Set up stage (digital): all the control registers (and write data register) are initialized
2. Execution stage (analog): perform the actual operation in the analog array
3. Read out stage (analog): convert the analog results into digital values
4. Addition stage (digital): perform the necessary operations for the integer matrix-matrix multiplication

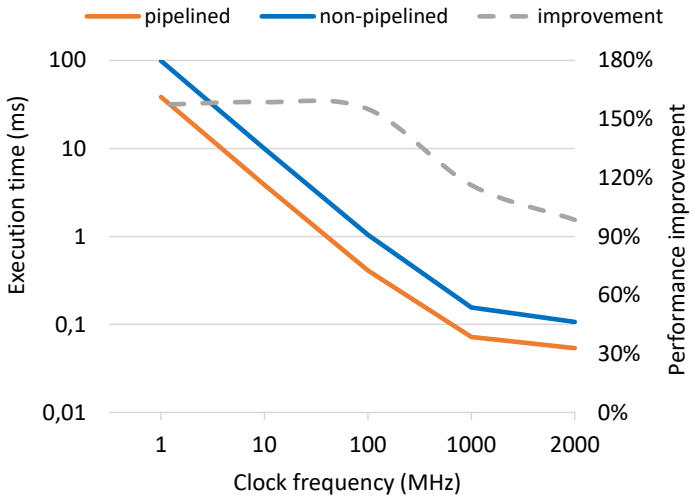

Figure 9: Performance improvement due to the unbalanced pipelining of tile used ReRAM/PCM device for GEMM benchmark

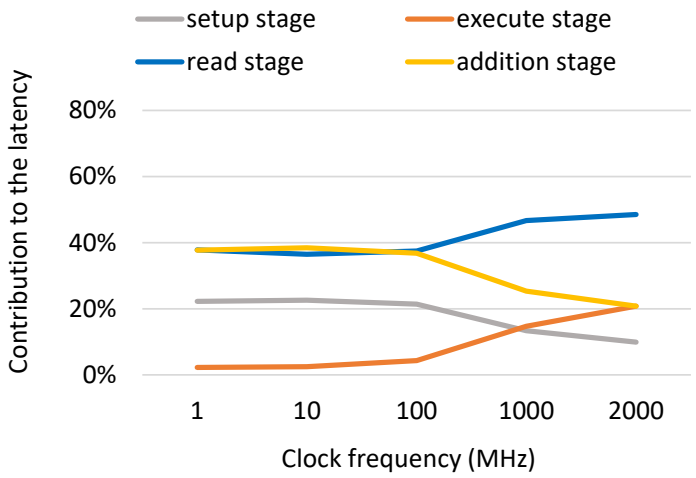

Figure 10: Contribution of each pipeline stage on the latency of the tile considering different clock frequencies

These stages sequentially follow each other while performing higher-level operations translated to a sequence of instructions in our ISA. It should be clear that the pipelining described here is different from the traditional instruction pipelining. In the latter, the latency of each stage should be matched with each other in order to have a balanced pipeline. In the CIM tile, the latency of the operation performed in the analog array is expected to be much longer than the latency of a single clock cycle in the digital periphery. Therefore, it is important that the right signaling is performed between the stages in order to enable pipelining. The introduced execution model to pipeline the operations within the CIM tile will allow for trade-off investigations between different NVM technologies and the (speed of the) digital periphery. Considering the aforementioned stages, the designer should evaluate the latency of each stage (which depends on the configuration of the tile, memristor technology, etc.) to realize the contribution of each stage to the total latency of the tile and merge some of them in the case there is a stage which has by far less latency than others. This analog/digital behavior of the CIM-tile restricts the choices and their effectiveness regarding the pipelining stages. It is worth mentioning that the two analog stages (*Execution* and *Read out* stages) cannot be split into more stages. This is the same for the *Set up Stage* as well since the registers initialized here cannot be changed before activating the crossbar to ensure the correct functionality of the system.

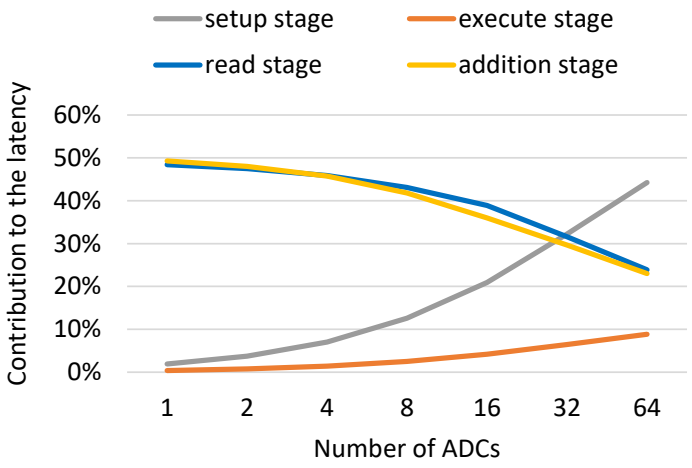

Figure 11: Effect of number of ADCs on the latency of pipeline stages in 100 MHz clock frequency

In the following, we provide an initial investigation of the CIM-Tile pipelining. First, we evaluate the effect of clock frequency in a CIM-Tile comprised of analog-digital circuits. The latency of the operations in the (analog) crossbar array is a constant number. Therefore, it is interesting to determine how fast the digital periphery should be clocked in order to 'match' this latency in order to make the pipeline more balanced. In the following investigation, we have fixed the number of ADCs to 16 and ran the GEMM benchmark at different frequencies. Figure 9 clearly shows that performance improvements can be gained by raising the frequency of the digital periphery. However, increasing the clock frequency beyond 1 GHz does not result in much better execution times as the analog circuits (relatively) are becoming the bottleneck. In addition, the performance improvement due to the pipelining will be reduced since the stages are more unbalanced. A positive side-effect is that pipelining more balanced stages will usually lead to better performance improvements over a non-pipelined design.

Second, Figure 10 depicts the latency breakdown consumed in each of the 4 stages against the clock frequency. Since several columns share an ADC, reading all the columns should be performed in multi-steps. In addition, after each ADC activation, digital processing is required in *'addition unit'*. Therefore, we can clearly observe that with a low frequency, the read and addition stages are completely dominant in the total latency. Using an efficient structure and minimum-sized adder, the latency of the addition unit is no longer than the read stage. With low clock frequency, the latency of the analog circuits is hidden in one clock period. However, As the clock frequency increases, the latency of the analog components starts to rise (relatively). Consequently, we can observe that the latency of the read stage, which is composed of analog (latency of ADC) and digital (decoding latency) latency, as well as the execute stage impose more latency. This information can be used to determine the number of pipeline stages for the actual implementation.

Third, Figure 11 depicts the latency breakdown in each of the 4 stages against the number of utilized ADCs. It should be clear that with increasing the number of ADCs, the number of cycles spent in the readout stage is greatly reduced. Consequently, we can observe that the relative contribution of the setup stage to the total latency grows accordingly. The contribution of the other stages to the total latency is almost negligible. With the advent of advanced ADC design to have more ADCs per tile, this figure brings insight into its implications and helps future design decisions.

4 Controller

This section describes the design of the tile controller. The controller serves two main purposes, namely 1) executing the nano-instructions from the compiled program, and 2) communicating with the outside unit when the WD/RD buffers need to be filled or the output buffer should be read. Figure 12 depicts the overview of the tile controller assuming two pipeline stages (by merging the first and second stages into one as well as the third and fourth stages into another).

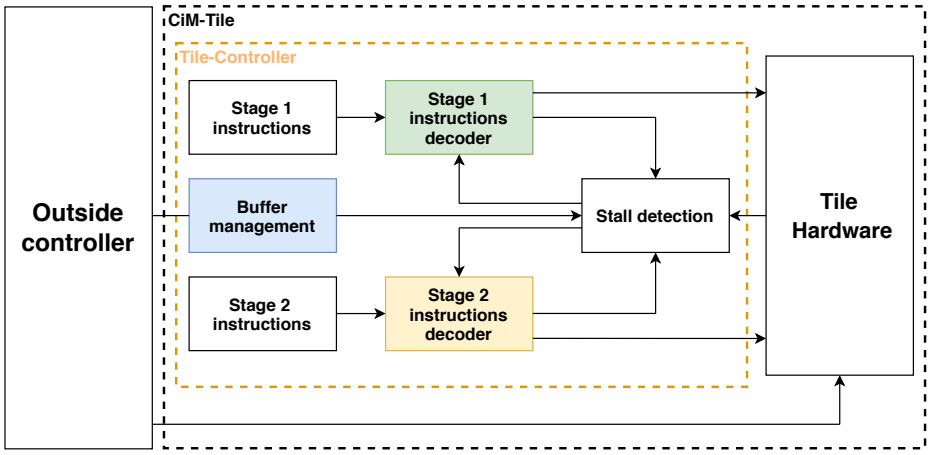

Figure 12: High-level CIM-tile controller to support two stages.

4.1 Instruction memory and decoder

As discussed before, despite traditional pipelining, here, each stage is associated with some instructions, and they are executed in parallel. Considering two pipeline stages, two instruction decoders are required. As the different stages correspond to separate, non-overlapping subsets of the ISA, the two decoders do not have to support all different instructions but just the instructions that correspond to their respective pipeline stage. This simplifies the hardware complexity of the decoder. Figure 13 shows how the instructions are stored in the instruction memory.

4.2 Stall detection

The CIM-Tile may consist of two or more pipeline stages. These stages depend on each other and have unbalanced latency. Hence, there should be a unit to synchronize these stages. The stall detection uses the information provided by the opcode of the instruction at the current PC of each stage to synchronize the stages to allow for proper program execution. Considering the two pipeline stage example, the first stage should be stalled in a few cases, such as 1) when valid data is not available in RD/WD buffer or 2) when the second stage is not ready to receive new data (still busy with reading from S&H), or 3) when stage 2 is performing a read for write verification. Similarly, the second stage should be stalled when 1) the first stage has not yet finished its operation instructions (indicated by DoS) or 2) a value should be written to the output buffer while the outside unit should still read the current values present in that buffer. The stall detection unit should be designed carefully to consider all possible scenarios.

4.3 Buffer management

The buffer management block sends signals to the stall detection and the outside unit providing information on whether there is valid data present in each of the buffers. The way the state of the buffers is monitored differs for each of the three buffers.

- Output buffer: The content of this buffer is valid after the execution of instructions that activate this buffer. This is either the CP or CB instruction for an architecture without or with the addition between the ADCs stage of the addition unit, respectively. For the read and logic operations, the output buffer is activated after the final read has been performed. The buffer management receives a signal from outside when the content of the buffer is read. Then it notifies the stall detection unit to allow for the next computation result can be stored in the buffer. This unit also sends a signal to the outside whenever the buffer is written with new information.

- WD buffer: We track the state of the WD buffer using a counter, indicating how many valid elements are present in the buffer. If there is no valid element in this buffer, the buffer management sets up a

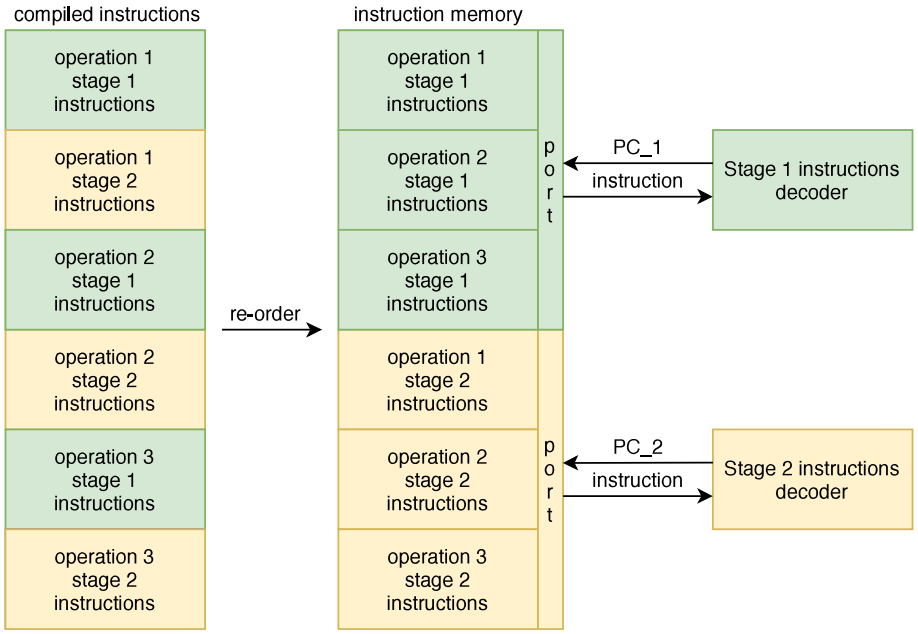

Figure 13: The overview of instruction memory.

flag to prevent the execution of any WD instruction. When new data is written into the WD buffer from the outside, the flag is reset.

- RD buffer: Controlling the RD buffer is more complex than other buffers. The number of valid bits in the RD buffer depends on the datatype size. The outside unit can provide a valid datatype size. Since there is an instruction to shift the buffer, the buffer management can explicitly deduce from the number of RDsh instructions that are executed when the buffer is empty.

4.4 Write verification

The write verification block only has to compare the crossbar output (stored in the read/logic register) to the data present in the WD register, setting the write-verify flag when one or more bits do not match. This comparison can be made by using a logical XOR gate. As only the output of the columns to which values were written should be verified, the comparison output is combined with the masking data from the WDS register through an AND gate. Finally, the logical OR operation can be used on the output produced for each of the columns to generate the verification flag, resulting in a logical '1' when a writing error has occurred. Figure 14 shows the implementation schematic per each crossbar column.

5 CIM Tile Compiler

The compiler translates high-level operations intended for the CIM-Tile into a sequence of instructions to be executed within the tile. The high-level operations (e.g., MMM) are provided by the front-end compiler which is responsible to search for the operations within the application program that can be performed using the memristor crossbar (see Figure 15). The front-end compiler receives the application in *Tensor Comprehensions* representation. This is then converted to a polyhedral representation in order to identify computational patterns suitable for acceleration by employing *Loop Tactics*. The front-end compiler is written by our partners in MNEMOSENE project and more information can be found in [DCZ+20]. Based on the requirements or constraints that come from either the tile architecture or technology side, our back-end compiler translates high-level operations to in-memory instructions. As depicted in Figure 15, this information is written to the configuration file and passed to the compiler. For example, there might be a constraint on the number of rows that can be activated

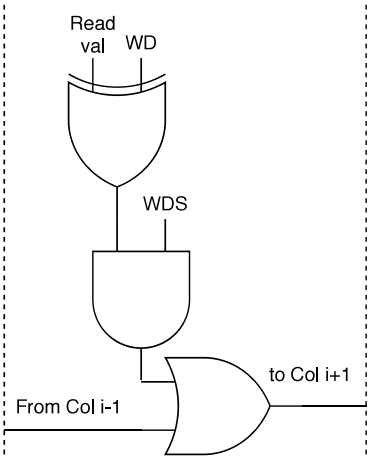

Figure 14: Implementation of write verification.

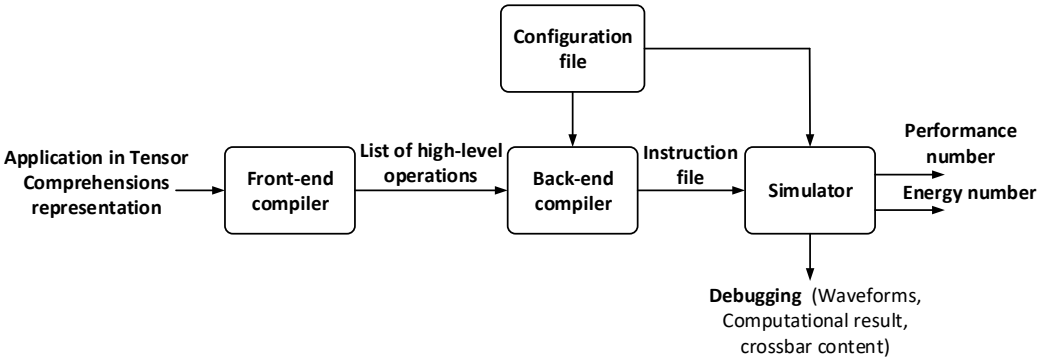

Figure 15: The overall system flow from tensor comprehensions down to fine-grained in-memory instructions

Table 5: Set of high-level operations and their semantics currently supported by the compiler.

Operation	Description	Semantic
MMM	Perform matrix-matrix multiplication between a matrix stored in the crossbar and an externally available matrix	MMM &A[d1][d2] - i - j - e - q - p - row - column
Store	Store a matrix at a specified location in the crossbar	Store &A[d1][d2] - i - j - p - q
Read	Read a matrix from a specified location from the crossbar	Read i - j - p - q
AND	Perform a logical AND operation on a specified set of rows and columns	AND v - j - q
OR	Performa logical OR operation on a specified set of rows and columns	OR v - j - q
XOR	Perform a logical XOR operation on a specified set of rows and columns	XOR v - j - q

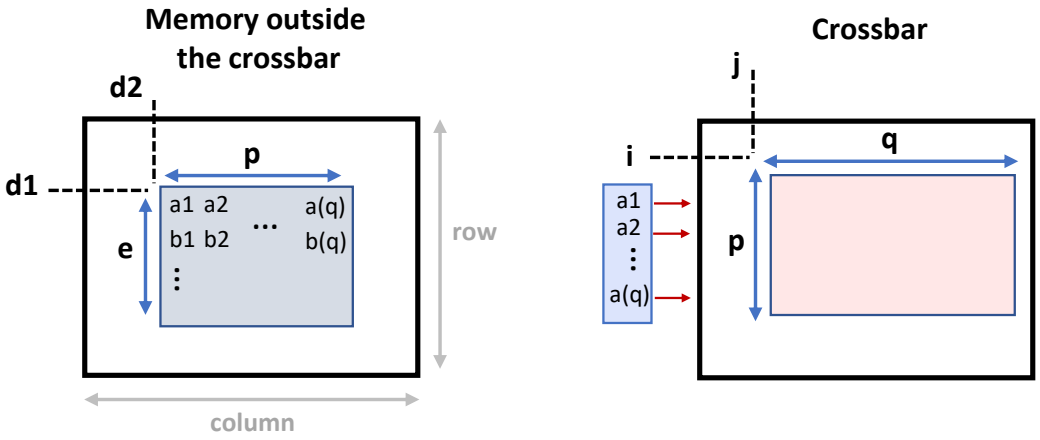

Figure 16: Illustration of parameters passed to the compiler for MMM operation.

at once. This constraint can come either from the precision of ADCs or even technology capability. Therefore, if an operation wants to activate more rows, the compiler splits it into several steps and takes care of other changes that might be needed. The flexibility brought by our in-memory instructions helps to overcome constraints, requirements, and sparse patterns. Figure 7 illustrates an example for VMM operation where four ADCs have to read columns in a special pattern (each 8 bit-lines share one ADC). This example demonstrates how the instructions deal with these kinds of patterns. It is important to note that the sequence of instructions generated by the compiler changes whenever the tile configuration changes. Therefore, by putting this complexity into the compiler, we try to keep the tile controller as simple as possible.

Table 5 lists the high-level operations and their parameters passed to the (back-end) compiler. The input data provided to the crossbar comes from a memory where its address is embedded into the parameters. The implementation of this is left for the future, where multi-tiling and communication among tiles and host is supported. In the current version, the compiler creates two files for RD and WD buffers where the CIM-Tile receives its input from outside. The compiler fills these two files with the correct data according to the address provided in the parameters of high-level operations. In the future, this data should come from other tiles or external storage.

To clarify more on the parameters passed into the high-level operations, Figure 16 depicts an example of MMM operation. In this operation, we assume the multiplicand matrix is already stored in the crossbar by a 'store' operation. The data of multiplayer is provided from memory outside the crossbar. The first parameter $\&A[d1][d2]$ provides the address to the first element of this matrix. Besides, 'i' and 'j' shows the coordination of the first element of multiplicand in the crossbar. 'P' represents the number of elements we have in one row of the multiplier, which should be equal to the number of elements in one column of the multiplicand. To clarify more, 'p' represents the number of active rows in the crossbar. It should be noted that the data from the multiplier matrix is provided to the crossbar row by row. Each row is copied to the RD buffer of the crossbar. We need to keep in mind each element may be represented by several bits. The RD buffer manages this for the crossbar.

Table 6: List of parameters used in the configuration file

crossbar	**drivers/analog peripheries**	**digital peripheries**
- number of rows/columns - cell levels	- number of ADCs - precision of ADCs	- clock frequency - datatype size
- cell resistances - cell read/write voltages	- ADC power - read/write drivers power	- energy per adder in addition unit
- write latency - read latency	- ADC latency per conversion - SH latency	- RS/WD/WDS/CS filling cycle - instruction decoding cycle - latency per adder

Finally, 'e' shows the number of rows in the multiplayer matrix, and 'q' represents the columns where the multiplicand matrix is stored. There are two other parameters that show the dimension of the external memory. This is required when we want to find the address of the first element in a new row in the multiplier matrix. However, for now, the compiler assumes a predefined size and ignores this information.

Similar to MMM, for the store operation, we identify the region in the crossbar where we want to write data by using 'i', 'j', 'p', and 'q' parameters. For now, we assume the data in the crossbar and outside memory are represented in the same way. We have to clarify that 'q' is the number of columns in the crossbar. If we store one bit per memristor and the datatype size is 8 bits, the number of elements we have in one row of crossbar or outside memory is $q/8$. Finally, for the logical operations, we want to activate two or more rows, and then, with a specialized sense amplifier, we compute the result of the operation. Based on what we observed from different applications, these activated rows (mainly two rows) are not necessarily next to each other and can be anywhere in the crossbar. In the current version, we pass a parameter 'v' to the compiler, which is a vector sized to the number of crossbar rows indicating which rows are selected.

6 CIM Tile Simulator

The proposed CIM architecture is generalized, making it capable of targeting different technologies with different configurations of the peripheral circuit. The simulator, written in SystemC, models the architecture presented and generates performance and energy numbers by executing applications. The simulator takes as input the program generated by the compiler, which is currently stored as simple (human-readable) text. In our HDL implementation, they are translated into binary bits. Besides the program, to simplify design space exploration, the configuration of architecture has to be sent to the simulator via a configuration file in which the user is able to specify many parameters. The simulator produces as output the following: (1) energy and performance numbers, (2) content of the crossbar (over time), (3) waveforms of all control signals, and (4) the computational results. All outputs are written into text files to be used for further evaluation. Figure 17 illustrates the control and data flow of the simulator considering just 2-stage pipelining. By decoding an instruction, the data embedded in it along with the control signals, are passed to the corresponding component related to that specific instruction in the data path. Then, the status will be returned to the controller to indicate the execution of the instruction is finished.

The simulator has been written in a modular way, which helps us to easily modify or replace the components shown in the tile architecture with new designs/circuits. Moreover, new attributes can be easily added to the components model, and their impacts are captured at the kernel level (e.g., read/write variability for the crossbar). In this fully parameterized simulator, each component has its own characteristics like energy, latency, and precision written into the configuration file. Table 6 shows all the parameters that can be set in the file to be used for the early-stage design space exploration. Furthermore, the first-of-its-kind feature of our simulator is the ability to calculate energy numbers according to the data provided by the application. Existing simulators estimate average energy numbers regardless of data. Our simulator considers the data stored in the array to estimate the energy consumption in the crossbar and its drivers. This is achieved by taking into account the data stored in the crossbar cell resistance level, the number of activated rows, and the equivalent resistance of the crossbar.

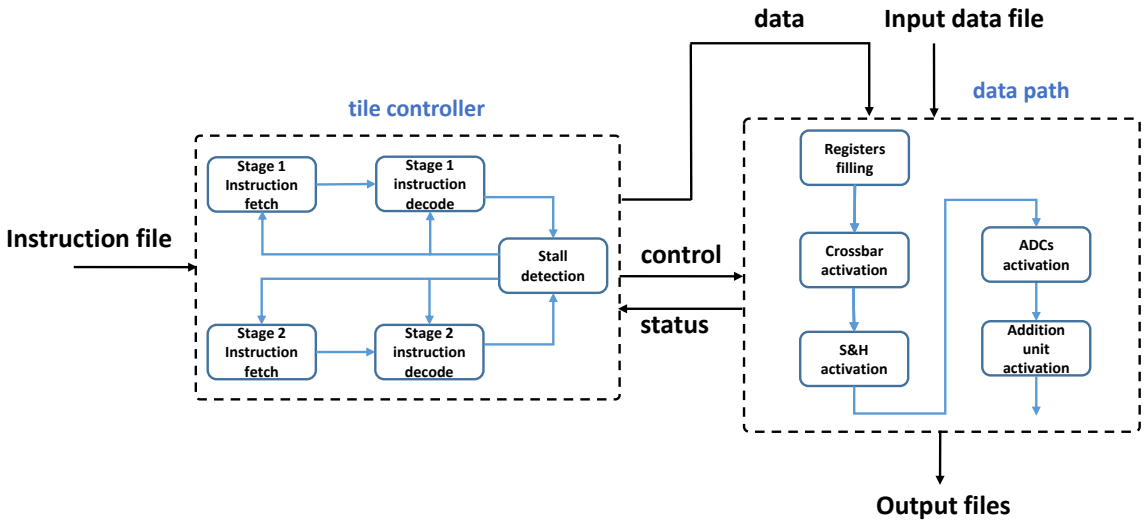

Figure 17: Simplified flowchart of the simulator comprising control flow and data path

The power consumption of the crossbar and read drivers regarding read/compute operations is given in Equation 1 where R_{rc} is the resistance level of the memristor cell located in row "r" and column "c", $P_{DIM_{read}}$ is the read drivers power, and $V(read)$ is the read voltages. In general, R_{rc} and $V(read)$ are members of two sets containing possible resistance and voltage levels, respectively. Furthermore, $activation_r$ is a binary value that indicates whether row "r" is activated and contributes to the power of the crossbar or not. Considering read and compute operations, the summation is performed for the selected rows and all the columns. In addition, for simplicity, the resistance of access transistors, as well as bit-lines, are ignored. The power consumption of write operations is shown in Equation 2 where $P_{DIM_{write}}$ is the write drivers' power. $V(write)$ and $I(write)$ are the write voltage and programming current, respectively. In general, $V(write)$ is a member of a set including different write voltage levels. Finally, $activation_c$ determines whether the column "c" is activated and would contribute to the crossbar energy or not. The summation is performed over the selected columns in just one activated row. The energy consumption of read/computational as well write operations are shown in Equations 3 and 4, respectively, in which $T_{Xbar(write)}$ as well as $T_{Xbar(read)}$ are the latency of the crossbar for write and read/computational operations. The latency of the crossbar for read/computational operations also depends on the peripheral circuits used to capture or read the analog values generated by the crossbar (S&H). Using S&H unit to capture the result, its capacitance is charged with different gradient according to the equivalent resistance of the crossbar. Therefore, the result should be captured at the right time when there is a maximum voltage difference on the capacitance of S&H unit for different crossbar equivalent resistances, which helps to be distinguished by the ADC easily (implies that the crossbar latency also depends on ADC capability). It is worth mentioning that the equations are data-dependent and provide the worst-case energy numbers for the crossbar and its drivers.

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} [(\sum_{c=1}^{\#columns} \frac{V^2(read)}{R_{rc}}) + P_{DIM_{read}}] * activation_r$$
$$R_{rc} \in \{L1, L2, ..., Ln\} \quad activation_r \in \{0, 1\}$$
$$V(read) \in \{V(r1), V(r2), ..., V(rn)\} \tag{1}$$

$$P_{(write)_r} = \sum_{c=1}^{\#columns} (V(write) * I(write) * activation_c + P_{DIM_{write}})$$
$$V(write) \in \{V(w1), V(w2), ..., V(wn)\} \tag{2}$$

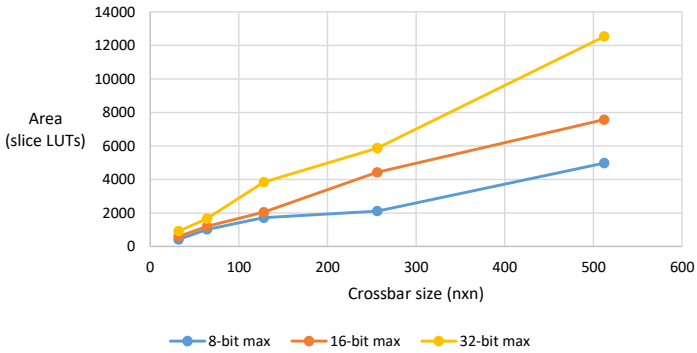

Figure 18: LUTs consumed by the digital CIM-Tile components.

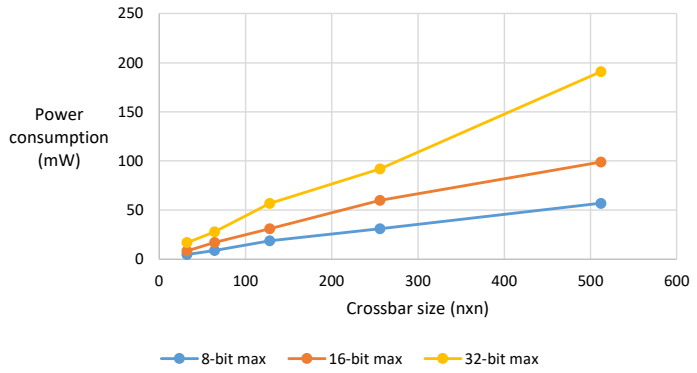

Figure 19: Power consumption of the digital CIM-Tile components implemented in FPGA.

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar(read,compute)} \tag{3}$$

$$E_{(write)} = P_{(write)} * T_{Xbar(write)} \tag{4}$$

7 FPGA Validation

Before any simulation, the functionality of the design should be verified first. To this end, we have implemented the CIM-Tile on the Xilinx Zynq ZC702 board, where analog components are modeled with digital circuits. In this side experiment, the correct execution of the instructions and the generated final output were evaluated to validate the functionality of the architecture. Similar to the simulator, the HDL has been written in a parametrized way. This also allows for exploring and directly comparing the characteristics of different configurations of the design. The following parameters can be swept in the HDL design: 1) crossbar size, 2) maximum datatype size, 3) the number of available ADCs (its implication on the 'Addition Unit'), 4) block size for the block-wise masking register filling, and 5) bus bandwidth to communicate with outside. It should be noted that the compiler generates the program, and this is stored in the FPGA before the execution. In the following, we provide the initial results of our FPGA implementation.

In theory, most digital components of the CIM-tile architecture, such as the buffers, registers, and the addition unit, scale linearly with the crossbar size as long as a constant ratio between the crossbar size and the number of ADCs is maintained. Figure 18 and 19 show the LUTs and power consumption reported by Vivado. In these reports, we only included the resources/power consumed

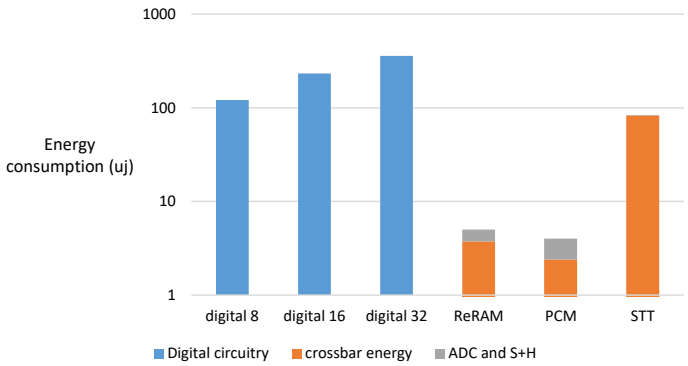

Figure 20: Energy comparison of Digital circuitry implemented in FPGA and analog components in 90 MHz clock frequency.

by the digital parts of the CIM-Tile (excluding the crossbar model, ADC, S&H, and Drivers). The simulation is done for three different datatype sizes, showing almost a linear increase in area usage and power consumption for each as we increase the crossbar size. In addition, the effect of supporting different datatypes can be seen. Clearly, as we increase the datatype size, more area and larger power are consumed due to mainly a larger RD buffer as well as an 'Addition Unit'.

Using the simulator, accurate numbers for energy consumption, taking actual data, and analog component activation into account can be obtained. The 'GEMM' benchmark is used to obtain these numbers. Figure 20 compares the energy consumption of the analog components and the digital circuitry for executing the benchmark at a clock frequency of 90MHz. It can be seen that the energy consumption for running the benchmark is significantly higher. This is because the analog components are not activated during the entire program. It should be kept in mind that the presented power and energy figures of the digital circuitry only represent the actual FPGA implementation of the circuitry. An ASIC implementation of the circuitry shows way lower power consumption, as seen in the following section.

8 ASIC Evaluation

Despite the previous section, we evaluate the ASIC implementation of the proposed architecture in terms of power, energy, and area compared with analog components. As a benchmark, the linear-algebra kernel "GEMM" from the Polybench benchmark suite was chosen. In this kernel, first, the multiplicands are written into the crossbar (write operation), and then the actual multiplication is performed. The values regarding the CIM-tile analog components are summarized in Table 7. The parameters for ReRAM and PCM technologies are taken from [HLTC+18] and [LG+18] validated for actual devices. The same ADC values used in [S+16] are considered for our setups. To obtain the power consumption and area for the CIM-tile digital circuits, they were synthesized in Cadence Genus targeting standard cell 15 nm Nangate library. Since all the information and control signals can be tracked using our simulator, a typical **activity factor** and performance number are extracted. These numbers are incorporated to obtain accurate energy consumption for the digital as well as analog components of the tile.

As we mentioned before, the energy consumption of the crossbar depends on the input and programming data. As more devices are programmed to LRS, and more rows are activated, clearly more energy is consumed during the computation. Since the simulator can actually execute kernels, the energy number obtained for the crossbar is data-dependent. Figure 21 depicts the energy consumption of the crossbar with different levels of sparsity. This figure illustrates the sparsity of logic value 1 for matrix-matrix multiplication. For this experiment, it is considered that the multiplier (input) and the multiplicand (programmed) matrix have the same sparsity. In addition, logic value 1 (0) as an input implies the corresponding row is activated (deactivated) and as programming data indicates, the memristive device is programmed to LRS (HRS). The simulation is performed for ReRAM and PCM devices, and an interesting observation is that although the ratio of HRS to LRS is lower in ReRAM

Table 7: Value of parameters used for the analog components.

Component	Parameters	Spec	
Memristive devices		ReRAM	PCM
	cell levels	2	2
	LRS	5k	20k
	HRS	1M	10M
	read voltage	0.2V	0.2V
	write voltage	2V	1V
	write current	100 μA	300 μA
	read time	10 ns	10 ns
	write time	100 ns	100 ns
Crossbar	structure	1T1R	
	num. columns	256	
	num. rows	256	
S&H	number	256	
	hold time	9.2 ms	
	latching energy	0.25 pJ	
	latency	0.6 ns	
DIM		read DIM	write DIM
	number	256	256
	power	3.9 μW	3.9 μW
ADC	power	2.6 mW	
	precision	8 bits	
	latency	1.2 GSps	

devices, since PCM devices have higher LRS compared to ReRAM, the sparsity leads to less variation in the energy consumption of the crossbar made with PCM.

Considering our CIM-tile architecture, digital circuits have different contributions to power consumption. Figure 22, depicted for 8-bit maximum supported datatype size in the tile, gives an insight into how much power each of these circuits consumes (8-bit datatype size means that in the case of MMM, each element of the multiplicand is distributed over 8 cells and each element of the multiplier is fed to the crossbar over 8 steps). As we expected, the *RD* buffer due to its size (feeding 8-bit data to 256 crossbar rows) and *addition unit* due to its high number of instances consume more power than others. However, since the buffer is not always switching (dynamic power), the average power is much less. The imposed power consumption on the system is the cost of the flexibility of row selection brought by the *RD* buffer and *RDS* register and their associate instructions.

The overhead of digital circuits in terms of energy is depicted in Figure 23 (a) and (b) to quantify the expense of high flexibility for programming the tile. In this figure, the overhead of the digital parts of our tile is compared with the analog parts. The comparison was performed for both ReRAM and PCM technologies with 256*256 crossbar size considering different datatype sizes. **First**, it is observed that as the datatype size increases, due to more computations, the overall energy is increasing as well. **Second**, by increasing the number of computations, the overhead of crossbar **pr**ogramming reduces (see striped blue bar) since more computations are performed before reprogramming the crossbar. **Third**, digital circuits impose more overhead while moving from 16-bit to 32-bit datatype size (see orange bars as well as energy per VMM). As the datatype size increases, fewer adders and registers must be instantiated (fewer numbers produced per crossbar activation). These units are essential to accomplish digital post-processing on the crossbar's output to deliver the final result. However, as the data type size increased, each of these adders and registers individually is of a larger size (see **??**). Accordingly, although this customized unit utilizes minimum-size adders and registers, supporting larger datatype size resulted in more overhead for this unit. **Fourth**, the overhead of the digital circuit for the PCM case is higher compared to ReRAM since the PCM crossbar consumes less energy due to its higher value of low resistance state. While the device level researchers are working on devices with a higher value of low resistance, this graph gives a good insight into how much this value contributes to the energy consumption of the system.

Finally, the area comparison of digital and analog circuits is depicted in Figure 24. For this experiment, the area of each cell in 1T1R crossbar structure is taken from [GAB+19], where it was fabricated with 22 nm technology. Although our digital circuits were synthesized with 15 nm technology, the result shows their area is around 6 times less than the analog counterpart, regardless of the crossbar dimension.

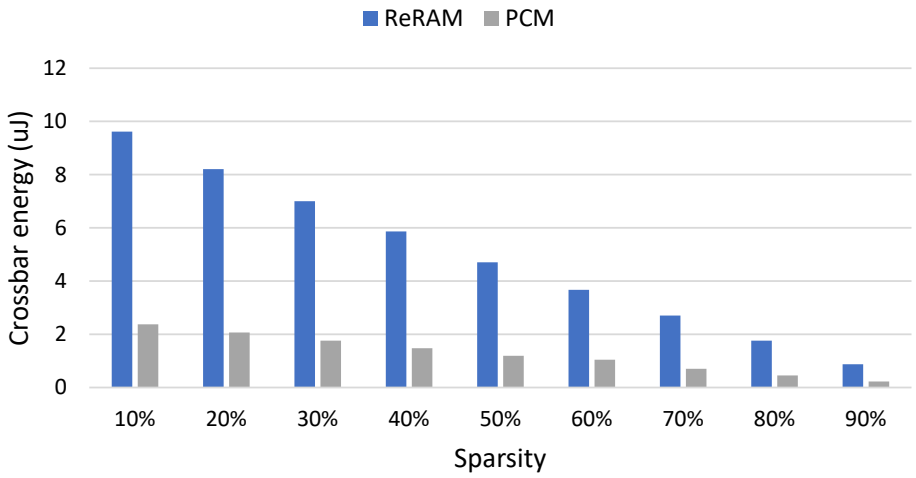

Figure 21: Energy consumption of the crossbar with respect to the percentage of sparsity for both input and programming data.

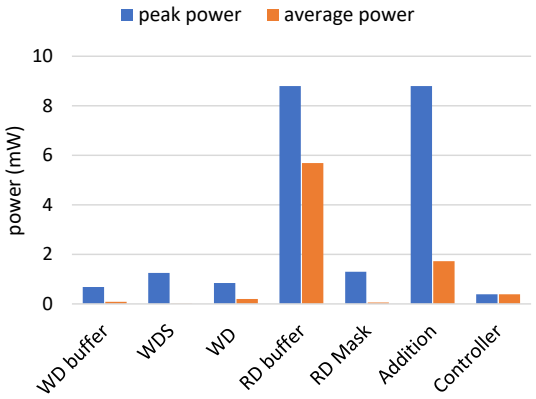

Figure 22: Power breakdown of digital circuits.

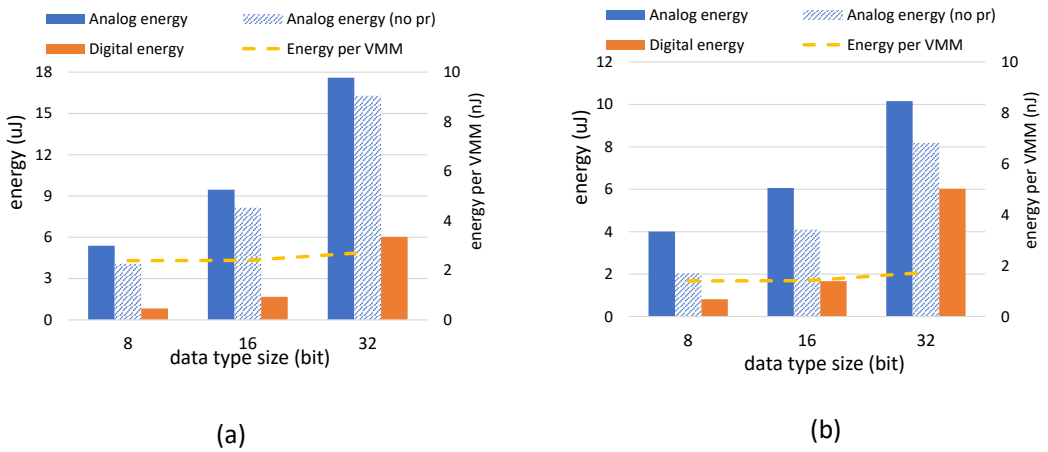

Figure 23: (a) Energy number for different datatype sizes considering ReRAM and (b) PCM with crossbar size of 256×256. It is assumed the entire crossbar contributes to the computation.

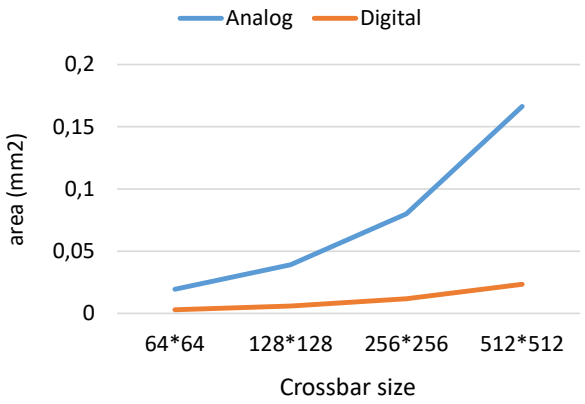

Figure 24: Area comparison of digital/analog circuits for ReRAM.

9 Conclusion

In this document, we presented our programmable CIM-tile architecture for which, by exploiting our ISA, high flexibility is achieved. In addition, the architecture provides a clear interface and independence from external devices. Considering different CIM-tile configurations, we developed our compiler and simulator to facilitate design space exploration. By synthesizing the digital part of the CIM-tile, its overhead compared to the analog counterpart was demonstrated in terms of power, energy, and area. The result will give a remarkable insight into future research to realize which customizations have to be applied for different configurations and application requirements.

References

[DCZ+20] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasilache. Tc-cim: Empowering tensor comprehensions for computing-in-memory. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[GAB+19] Oleg Golonzka, U Arslan, P Bai, M Bohr, O Baykan, Y Chang, A Chaudhari, A Chen, J Clarke, C Connor, et al. Non-volatile RRAM embedded into 22FFL FinFET technology. In *2019 Symposium on VLSI Technology*, pages T230–T231. IEEE, 2019.

[HLTC+18] Alexander Hardtdegen, Camilla La Torre, Felix Cüppers, Stephan Menzel, Rainer Waser, and Susanne Hoffmann-Eifert. Improved switching stability and the effect of an internal series resistor in HfO 2/TiO x bilayer ReRAM cells. *IEEE transactions on electron devices*, 65(8):3229–3236, 2018.

[LG+18] Manuel Le Gallo et al. Compressed sensing with approximate message passing using in-memory computing. *IEEE Transactions on Electron Devices*, 65(10):4304–4312, 2018.

[S+16] Ali Shafiee et al. ISAAC: A convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.